dsPIC Threads Library 1.0

Dean Ferreyra
dean@octw.com
http://www.bourbonstreetsoftware.com/

November 14, 2008

Contents

Chapter 1

Introduction

The dsPIC Threads Library provides basic cooperative multitasking/multithreading to the Microchip dsPIC family of microcontrollers. It is written mostly in C. It implements a simple round-robin style task switcher.

The cooperative multitasking design results in task switches that are less expensive than would be possible with preemptive multitasking. This also allows the library to be implemented in such a way that interrupts are never disabled.

This library provides basic thread start and stop functions, a way to associate thread-local storage to threads, and some basic thread management functions, including a way to determine a thread's stack usage.

I have released this code under the GNU General Public License with the hope that others might find it useful.

Chapter 2

Building From Sources

To build the dsPIC Threads Library, you need to have already installed the Microchip MPLAB C30 cross-compiler. You also need a basic Unix-like build environment with programs like Make and Sed. Under Windows I personally install and use Cygwin.

This library also depends on the dsPIC Helper Library that can be found at `http://www.bourbonstreetsoftware.com/DsPICDevelopment.html`. After building that library, you will have to ensure that its header files are available to the dsPIC Threads Library. The `Makefile` in the `src` directory has a variable called `DSPIC_HELPER_DIR` that you can edit to specify the location of the dsPIC Helper Library directory tree.

To build the dsPIC Threads Library, first unpack the source tree. You can do this with the following command:

```
$ tar -xzf dspic-threads-1.0-src.tgz
```

Next, change directory to the head of the source tree that was just unpacked and run the `make` command:

```
$ cd dspic-threads-1.0
$ make
```

This should build the dsPIC Threads Library for the supported dsPIC microcontrollers. This includes the dsPIC30F6015, dsPIC30F6010A, and the dsPIC30F6010, though you can easily modify the `Makefile` to build for other targets as well.

2.1 Library Types

The build process actually builds two libraries for each microcontroller type; one for the default code size, and one using the `-mlarge-code` option. The libraries are built in separate directories named after the corresponding microcontroller and which option was enabled. For example, for the dsPIC30F6015, the libraries can be found here:

- src/30F6015/libdspic-threads.a
- src/30F6015-large-code/libdspic-threads.a

Chapter 3

Working With the Library

3.1 Initializing the Library

Functions

`void threads_init(void)` ***function***

This function initializes the library and must be called before calling any other library function. This also designates the current execution state as the primordial thread.

3.2 Starting and Stopping Threads

Functions

`void threads_start(volatile thread_data_t* thread, void (*fn)(void), uint16_t* stack, uint16_t stack_size)` ***function***

This function schedules a new thread and sets that thread to the "runnable" state. The caller needs to provide the space for `thread` and `stack`. The `stack_size` argument is the size of `stack`, in words. The stack space provided is initialized with a particular bit pattern to help detect stack usaged.

The new thread is scheduled to follow the current thread. The thread starts life by calling `fn`. If `fn` returns, the thread is unscheduled and an implicit task switch occurs.

The call to `threads_start()` does not force a task switch.

`bool threads_stop_current(void)` ***function***

This function unschedules the current thread, even if that thread is the primordial thread. The only limitation is that the last thread in the queue cannot be unscheduled. If the current thread cannot be unscheduled, `threads_stop_current()` returns `false`.

The call to `threads_stop_current()` does not force a task switch, so a task switch must be explicitly performed to move on to the next thread. For example, the following pattern can be used when stopping the current thread:

```
// Try to stop the thread
if (threads_stop_current()) {
  // Stop succeeded, now task switch
  threads_task_switch();
  // Will never get here
} else {
  // Will get here if stop failed
}
```

`bool threads_stop(volatile thread_data_t* thread)` ***function***

This function unschedules the given thread, even if that thread is the primordial thread. The only limitation is that the last thread in the queue cannot be unscheduled. If `thread` cannot be unscheduled, `threads_stop()` returns `false`.

3.3 Task Switching

This library uses a cooperative multitasking scheme, so all threads must explicitly request task switches to allow other threads to run.

Functions

`void threads_task_switch(void)` ***function***

This function forces a task switch.

Because the library uses cooperative multitasking, this function must be called to allow another thread to run. For smooth multitasking, you should arrange your code so that each thread calls this function periodically.

There is only one situation that leads to an implict task switch: If the thread function passed to `threads_start()` ever returns, the thread is unscheduled and an implicit task switch is performed.

3.4 Thread-local Storage

Functions and Macros

`void threads_tls_set(volatile thread_data_t* thread, void* tls)` ***function***

This function associates pointer `tls` with the given `thread`. This pointer can be retrieved by using `threads_tls_get()` or `threads_tls_current_get()`.

`void* threads_tls_get(volatile thread_data_t* thread)` ***macro***

This function retrieves the pointer associated by `threads_tls_set()` with the given `thread`.

`void* threads_tls_current_get(void)` ***macro***

This function retrieves the pointer associated by `threads_tls_set()` with the current thread.

3.5 Thread Management

Functions and Macros

`thread_data_t* threads_current_get(void)` ***macro***

This macro retrieves a pointer to the internal thread data for the currently running thread. This pointer can be passed to any of the functions requiring a `thread_data_t` pointer.

`uint16_t threads_stack_usage(const volatile thread_data_t* thread)` ***function***

This function returns the number of stack words that have been used by `thread`. If `thread` is the primordial thread, this function returns `0`.

`void threads_runnable(volatile thread_data_t* thread, bool run_p)` ***macro***

This macro sets the "runnable" state to `run_p` for `thread`. Only threads in the "runnable" state will be considered during a task switch. Note that if no thread is in the "runnable" state, a call to `threads_task_switch()` will result in an infinte loop.

Chapter 4

Example Code

Here is a simple example program that initializes the library and then starts two threads. The primordial thread plus the two new threads flash LEDs at different rates. All the threads call the example `sleep()` function which ensures that all the threads will run.

```
#include <p30fxxxx.h>
#include <threads.h>

// Hardware-specific configuration

_FOSC(CSW_FSCM_OFF & XT_PLL8);
_FWDT(WDT_OFF);
_FBORPOR(PBOR_ON & BORV_20 & PWRT_64 & MCLR_EN);

// Hardware-specific port specifications

#define LED_0_TRIS() (_TRISA9 = 0)
#define LED_0_OFF()  (_LATA9 = 0)
#define LED_0_ON()   (_LATA9 = 1)

#define LED_1_TRIS() (_TRISA10 = 0)
#define LED_1_OFF()  (_LATA10 = 0)
#define LED_1_ON()   (_LATA10 = 1)

#define LED_2_TRIS() (_TRISA14 = 0)
#define LED_2_OFF()  (_LATA14 = 0)
#define LED_2_ON()   (_LATA14 = 1)

// Sleep routine
void sleep(uint16_t n)
{
```

```
    for (uint16_t i = 0; i < n; i++) {
        for (volatile uint16_t j = 0; j < 1000; j++)
            ;
        threads_task_switch();
    }
}

#define STACK_SIZE 64

// Thread 1 configuration

thread_data_t thread1_data;
uint16_t thread1_stack[STACK_SIZE];

void thread1(void)
{
    // Flash LED 1 forever
    for (int i = 0; i < 5; i++) {
        LED_1_ON();
        sleep(50);
        LED_1_OFF();
        sleep(50);
    }
}

// Thread 2 configuration

thread_data_t thread2_data;
uint16_t thread2_stack[STACK_SIZE];

void thread2(void)
{
    // Flash LED 2 forever
    for (int i = 0; i < 5; i++) {
        LED_2_ON();
        sleep(200);
        LED_2_OFF();
        sleep(200);
    }
}

int main(void)
{
    // Initialize some LED ports as output ports
    LED_0_TRIS();
    LED_1_TRIS();
```

```
    LED_2_TRIS();

    // Initialize threads library
    threads_init();

    // Start two threads
    threads_start(&thread1_data, thread1,
                  thread1_stack, STACK_SIZE);
    threads_start(&thread2_data, thread2,
                  thread2_stack, STACK_SIZE);

    // Primordial thread flashes LED 0 forever
    for (;;) {
        LED_0_ON();
        sleep(100);
        LED_0_OFF();
        sleep(100);
    }
}
```

Index